

Object-Oriented Micro-Review

OOP *What?*

Marc A. Murison

Astronomical Applications

U. S. Naval Observatory

Washington, D.C.

28 April, 1997

Purpose of this Talk

- Quick overview of object-oriented concepts
- Definitions of common terms
- Overview of design cycle of an object-oriented project
- Show a few simple examples from Newcomb

Advantages of OOD/OOP

- Models the real world more closely
 - Deals with
 - objects
 - interactions between objects
 - Programs are easier to understand and maintain
- Flexibility and **extensibility**
- Data **encapsulation**
 - The only way to interact with object data is through a public interface
 - Data is "protected" from arbitrary access
 - As opposed to fortran common blocks or C structures
- Code and architecture **reuse**
 - Object **inheritance**
- **Polymorphism** (virtual functions)

Terminology

- **ADT** = Abstract Data Type
 - You're used to "regular" data types (RDT)
 - **double**
 - **int**
 - RDTs consist of a public representation (double, int, etc.) and a public set of operations (addition, multiplication, etc.)
 - ADTs are the generalization of this
 - **String**
 - **Vector**
 - **Observation**
 - ADTs consist of
 - a **private** representation (collection of *data members*), which defines the object properties
 - a **public** set of operations (*member functions*), which defines the object interface

Terminology (continued)

● Class

- C++ classes allow users to define their own ADTs.

```
▪ class BigInt {                                class declaration
  public:                                       access level
    BigInt( int, int );                          constructor
    ~BigInt();                                    destructor
    BigInt mul (BigInt, BigInt);                member function
    ...
  private:
    int int1, int2;                            private data
  };
```

● Overloading = giving two or more functions the same name

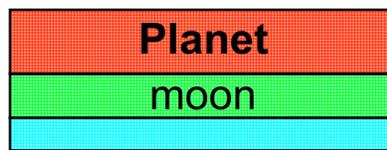
- `int max(int, int)`
- `double max(double, double)`
- Compiler gets it straight because of *name mangling*
 - function name
 - function argument types
 - return object type

Terminology (continued)

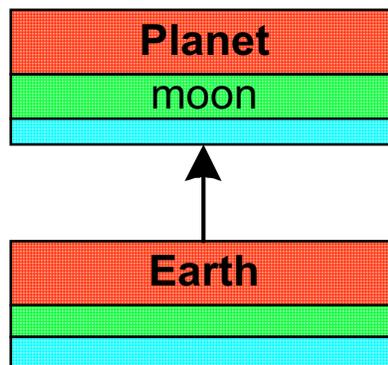
● Inheritance

- most important mechanism for code reuse
- Object-oriented way of implementing an "is a" relationship between objects
- Example

- a Planet "has a" moon



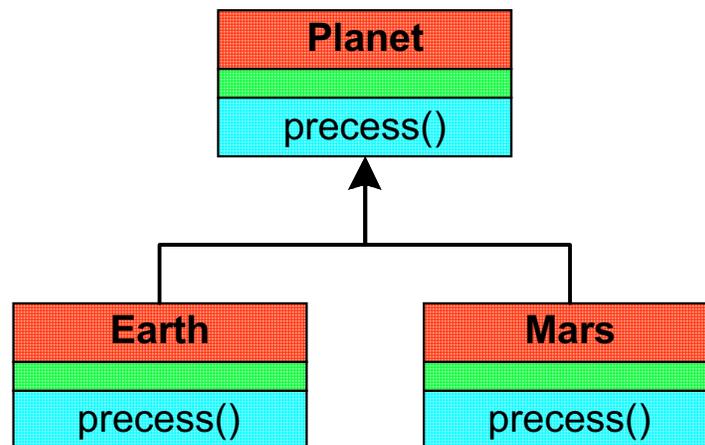
- but the Earth "is a" planet



- A derived class (aka subclass) inherits all the data and functionality of its parent class
 - Only additional data and functionality specific to derived class needs to be coded

Terminology (continued)

- **Multiple Inheritance** = simultaneously inherit from more than one parent class
- **Virtual Functions**
(aka **polymorphism** or **dynamic binding**)
 - linking is done *dynamically* at runtime
 - virtual function tables
 - Example



```
Planet *p;  
...  
p = new Earth;  
...  
p->precess ();           calls Earth::precess()
```

Terminology (continued)

● Templates

- mechanism for creating generic ADTs
- furthers code reuse
- Example of a template function

```
template<class T>
T max( T x, T y );
...
double a, b, c;
int    x, y, z;
Vector A, B, C;
...
c = max( a, b );
z = max( x, y );
C = max( A, B );
```

- Example of a template class

```
Planet          p, q, r;
Stack<Planet>   sp;
...
sp.push( p );
...
```

Example from Utilities: Stack

```
template< class T >
class Stack {
public:
    Stack( void ) : top(NULL), nelem(0) { }
    Stack( const Stack<T> &S );
    virtual ~Stack( void ) { clear(); }

    Stack<T> & operator = ( const Stack<T> &S );

    void push( const T &x ) throw(Error);
    T      pop( void ) throw(Error);
    void clear( void );

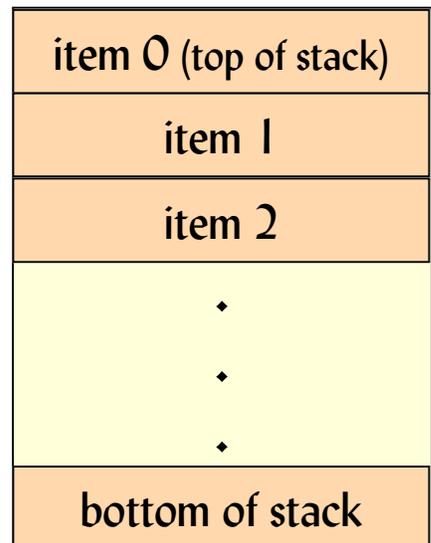
    inline Boolean isempty( void ) const;
    inline uint size( void ) const;

protected:
    class StackNode {
    public:
        StackNode( const T &x ) : nextnode(NULL), data(x) { }
        ~StackNode( void ) { }

        StackNode * nextnode;
        T          data;
    };

    StackNode * top;
    uint      nelem;

private:
    void copy( const Stack<T> &S );
};
```



Terminology (continued)

- **Exception Handling**

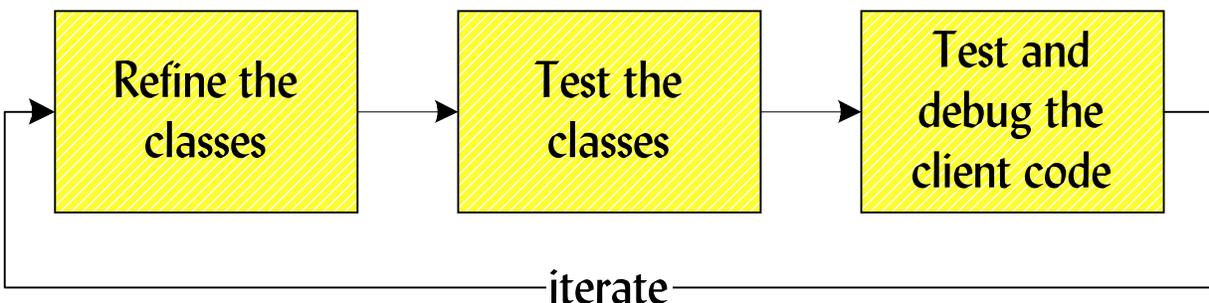
- allows *graceful* handling of exceptional conditions
- Example

```
try {  
    func1 ();  
    func2 ();  
};  
catch( Error &e ) {  
    cout << "You're screwed, dude!"  
        << e.why ();  
    cleanup ();  
    exit(FAIL);  
};  
func3 ();  
...
```

- **RTTI** = RunTime Type Identification
- **OOP** = Object-Oriented Programming
- **OOD** = Object-Oriented Design
- **OOA** = Object-Oriented Analysis

Object-Oriented Design Cycle

- Identify the **classes**
 - Write a description of the problem — in words.
 - Nouns = candidates for **objects** (classes)
 - Verbs = candidates for **methods** (class functions)
 - Identify the class relationships
 - **is-a**: subclass (derived class)
 - **has-a**: encapsulated data
- Define the interface and the class properties
- **Test each class with a separate test program**
- Implement the classes in the client code
- Define new classes and refine the existing classes as needed



Example from Newcomb: Observation

Observation	
Point<double>	data, errs
Point<Boolean>	objexists
JulDate	jd
ObsType	otype
JulDate	JD()
ObsType	type()
Point<double>	coords()
Point<double>	errors()
Point<Boolean>	exists()

Example from Newcomb: Observation

```
class Observation {
public:
    enum {equatorial, horizon,
          delayDoppler, VLBI, occult} ObsType;

    Observation( const JulDate &juldat,
                 const Point<double> obs,
                 const Point<double> err,
                 const ObsType ot,
                 const Point<Boolean> exist );
    Observation( const Observation &obs );
    ~Observation( void ) {}

    Observation & operator = ( const Observation &obs );

    // Extract data from the Observation
    JulDate      JD      ( void ) const;
    ObsType      type    ( void ) const;
    Point<double> coords( void ) const;
    Point<double> errors( void ) const;
    Point<Boolean> exists( void ) const;

protected:
    Point<double> data, errs;
    Point<Boolean> oexists;
    JulDate      jd;
    ObsType      otype;
};
```

Example from Newcomb: ObservationSet

ObservationSet	
Set<Observation>	data
String	objname
void	add(Observation)
void	add(ObservationSet)

Example from Newcomb: ObservationSet

```
class ObservationSet {  
  
    friend class ObservationSetIterator;  
  
public:  
    ObservationSet( const String name,  
                   Set<Observation> obs );  
    ObservationSet( const ObservationSet &o );  
    ~ObservationSet( void );  
  
    ObservationSet &  
        operator = ( const ObservationSet &o );  
  
    // Add new data to the ObservationSet  
    void add( const Observation newobs );  
    void add( const Set<Observation> &newobs );  
  
protected:  
    Set<Observation> data;  
    String          objname;  
};
```

Example from Newcomb: ObservationGroup

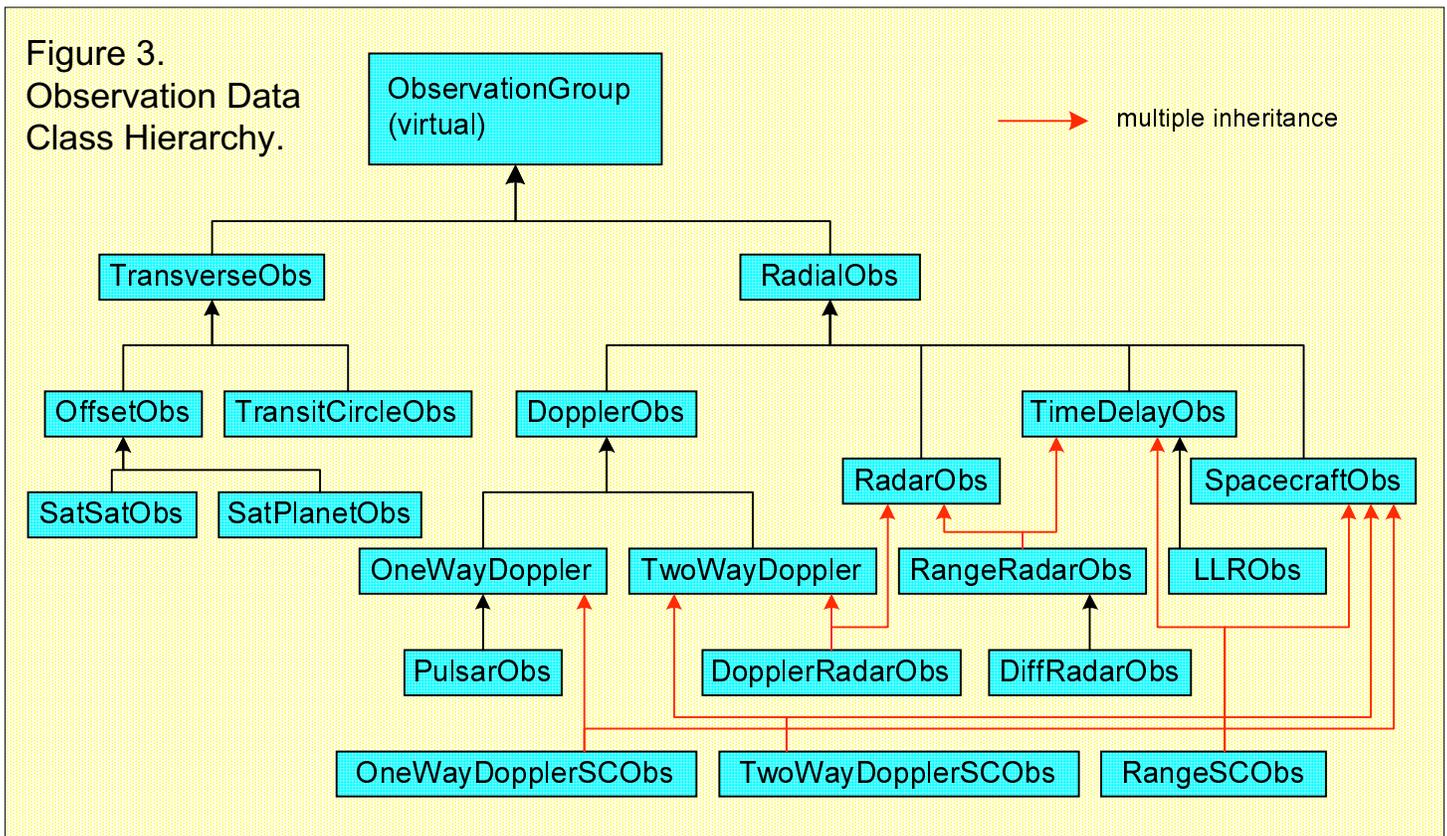
ObservationGroup	
Set<ObservationSet>	data
String	groupname
Observatory	place
String	name()
uint	size()
void	add(String objname, Observation obs)
void	add(ObservationSet)

Example from Newcomb: ObservationGroup

```
class ObservationGroup {  
  
    friend class ObservationGroupIterator;  
  
public:  
    ObservationGroup( const String &name );  
    ObservationGroup( const ObservationGroup &c );  
    virtual ~ObservationGroup( void );  
  
    ObservationGroup &  
        operator = ( const ObservationGroup &c );  
  
    String name( void ) const {return groupname;}  
    uint size( void ) const {return data.size();}  
  
    // add new data to the observation group  
    void add( const String &objname,  
             const Observation &obs );  
    void add( const ObservationSet &obsset );  
  
protected:  
    Set<ObservationSet> data;  
    String groupname;  
    Observatory place;  
};
```

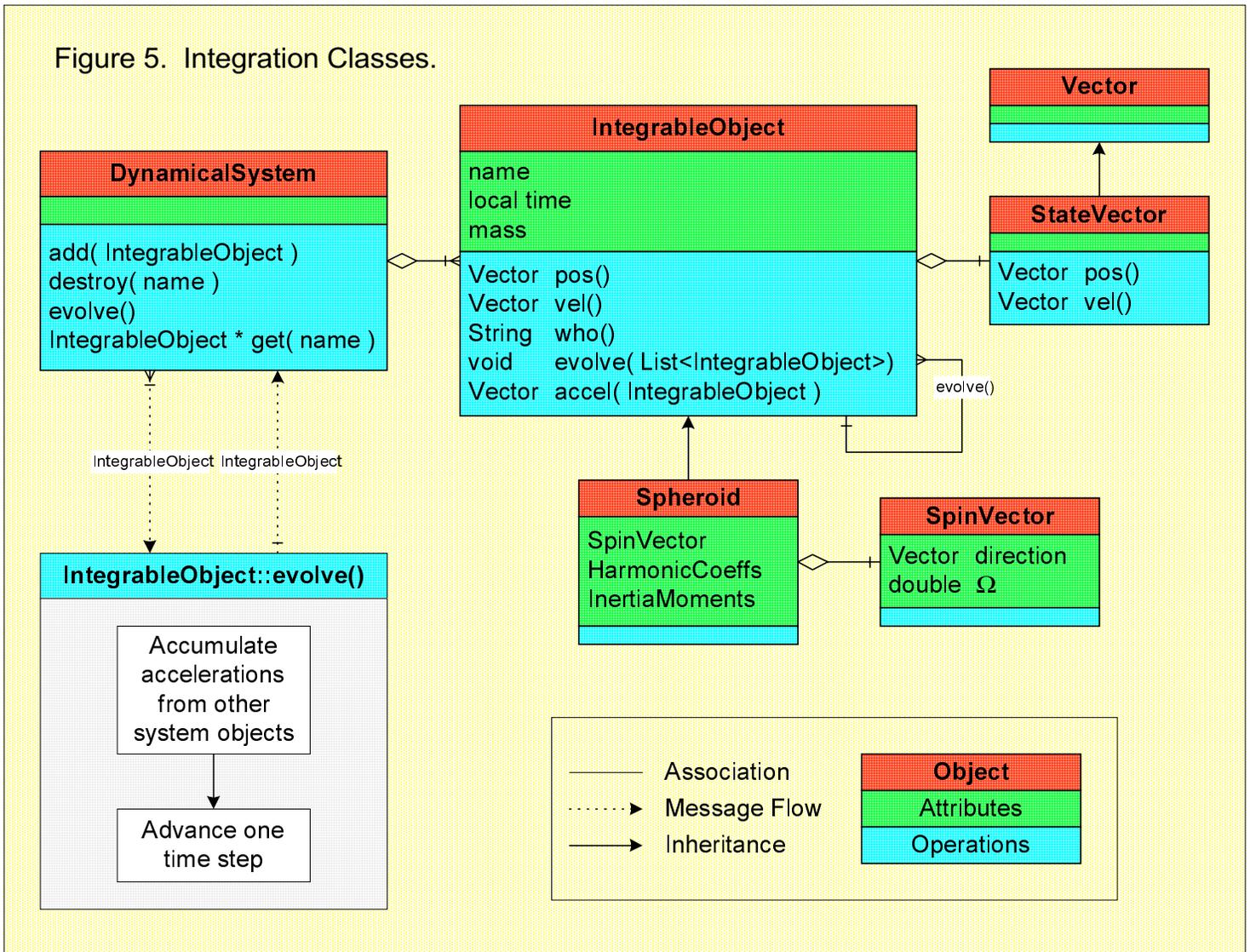
Example from Newcomb: ObservationGroup

Figure 3.
Observation Data
Class Hierarchy.



Example from Newcomb: Integration Classes

Figure 5. Integration Classes.



Merging Design and Source Code Generation

- Tools now exist that allow you to generate source code from a design (and *vice versa*)

The screenshot displays the Borland C++ IDE. The main window is titled "Unified Object Model Editor [sam...]" and shows a class diagram. The diagram includes an abstract class "AbstractClass" with a destructor "+~AbstractClass ()". Below it is a concrete class "Class" with attributes "+attrPublic : char" and "-attrPrivate : short", and methods "+operVirtual () : void", "+operPublic () : int", "-operPrivate () : void", and "-operConst () : void". A "Whole" class is shown with an aggregation relationship to "Class".

To the right, a source code window titled "C:\...\CLASS.H" shows the corresponding C++ code:

```
#include "PART.H"
#include "UTILCLSS.H"

class Class : public Abstrac
public:

    char attrPublic;
    virtual void operVirtual(
    int operPublic());
private:

    short attrPrivate;

    void operPrivate();
    void operConst() const;
    /**# :[Label = "aggregatic
        [Qualifiers = "Whole
    vector <Part> mPart_;
    UtilityClass * miUtilityCl
};
#endif
```